Comparing a Commercial and an SDN-Based Load Balancer in a Campus Network

by

Ashkan Ghaffarinejad

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2015 by the
Graduate Supervisory Committee:

Violet R. Syrotiuk, Chair
Guoliang Xue
Dijiang Huang

ARIZONA STATE UNIVERSITY

May 2015

ABSTRACT

Commercial load balancers are often in use, and the production network at Arizona State University (ASU) is no exception. However, because the load balancer uses IP addresses, the solution does not apply to all applications. One such application is Rsyslog. This software processes syslog packets and stores them in files. The loss rate of incoming log packets is high due to the incoming rate of the data. The Rsyslog servers are overwhelmed by the continuous data stream. To solve this problem a software defined networking (SDN) based load balancer is designed to perform a transport-level load balancing over the incoming load to Rsyslog servers. In this solution the load is forwarded to one Rsyslog server at a time, according to one of a Round-Robin, Random, or Load-Based policy. This gives time to other servers to process the data they have received and prevent them from being overwhelmed. The evaluation of the proposed solution is conducted a physical testbed with the same data feed as the commercial solution. The results suggest that the SDN-based load balancer is competitive with the commercial load balancer. Replacing the software OpenFlow switch with a hardware switch is likely to further improve the results.

To my mother and stepfather

ACKNOWLEDGEMENTS

I would like to thank

my advisor Dr. Violet R. Syrotiuk for all the help and support she gave me towards getting my master's, and who taught me how to do research. This thesis would not have happened without the endless help of Chris Kurts, *"The Splunk Guy"* as he introduces himself.

Thanks to Jack Hsu, for accepting me into his team and providing us the problem and the testbed that we needed. We appreciate Patricia Schneider, our project manager who worked with us and UTO to solve problems and keep the project moving forward. Thanks to Jay Steed, for giving us this amazing opportunity to do our research in SDN using ASU's production network and resources. Thanks to Dr. Guoliang Xue and Dr. Dijiang Huang for serving on my committee.

Finally, I would like to thank my mother and my stepfather for always being there for me and supporting me without boundaries through all these years. Without their help I would not have made it to this point. You are always in my heart.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

1.1 Motivation

With the rapid growth of web applications in the late 1990's, load balancing was used to divide the network load among identical web servers in order to minimize the service time to users and maximize the performance of servers. At that time, techniques such as those based on DNS and adaptive TTL were exploited by enterprise administrators [8].

Today, commercial load balancers are often in use, including in the production network at Arizona State University (ASU). But sometimes there are applications that are not well suited to how the commercial products balance the load. One such application is Rsyslog. This application is in charge of processing syslog packets and writing them into files. It receives its input from Palo Alto firewalls. This firewall generates a wide variety of log messages to alert the network administrator of an existing issue in or a threat to the entire campus network. As can be imagined the amount and the speed of data that this application generates is huge. The current load balancing solution at ASU is used to spread the load among several Rsyslog servers. The problem is that the existing load balancer does not divide the load equally among the servers because it uses the source IP address to divide the load. Also, it cannot forward the load to one Rsyslog server at a time because there are multiple Palo Alto firewalls and each one of them is mapped to an Rsyslog server. This leads to having unequal sized files stored on the servers as well as the Rsyslog servers being overwhelmed with the amount of input and losing data. To address

this problem, we propose to spread the load among servers using a solution based on software defined networking (SDN).

1.2 Contributions

We propose a solution to load balance the data coming from a Palo Alto firewall among servers using software defined networking. This is accomplished by placing an OpenFlow switch in front of the incoming data. The OpenFlow switch, controlled by a controller that we have developed, gives us the ability to forward these packets to our desired server. This solution enables us to make changes to forwarding as circumstances in the network change.

We develop three different load balancing policies: Round-Robin, Random and Load-Based. The first two policies work without measuring forwarded load to each server. However, in the Load-Based policy our goal is to be aware of the amount of forwarded load to each server in order to spread the load as equally as possible. We use a software switch to run our experiments in a testbed using a real data feed from a Palo Alto firewall.

We evaluate each load-balancing policy, both with and without server failures, for a period of five days. We compare the results with syslog data stored on files at our servers to that generated by ASU's current commercial load-balancing solution. Given that the Palo Alto feed uses UDP, which may result in packet losses, we find that our solution, regardless of the policy in use and regardless of having server failures or not, is able to deliver the data at a rate (computed by dividing number of syslog logs stored on the testbed to the number of syslog logs stored on ASU's Rsyslog servers) with the commercial solution. As the results of experiments show, the data delivery rate is in the range of 0.9923 to 1.0446 with the mean of 1.0109 and the standard deviation of 0.0197. (A rate of over 1 is possible because, while the feeds from the

Palo Alto are the same, the path internal to ASU's network is different and losses occur from the use of UDP.)

To evaluate which load-balancing policy spreads the load more equally among the servers, we define a split ratio as the ratio of the smallest sized file by the largest sized file received on our servers. The closer the split ratio is to one, the better the load-balancing policy divides the load. In the first experiment, without having server failures, the average split ratios over five days period for the Round-Robin, Random and Load-Based policies are 0.9906, 0.9695 and 0.9809, respectively. In the second experiment, where servers may fail, the average split ratios for the Round-Robin, Random and Load-Based policies are 0.8298, 0.8791 and 0.9796, respectively.

Our results show that Round-Robin policy divides the load slightly better than the other two policies when there is no server failures. However, in the second experiment, where server failures exist, the results show us that the Load-Based policy is the best choice by far. Overall the Load-Based policy has the best split ratio average over the two experiments.

We conclude our software defined networking load-balancing solution is competitive with the existing commercial solutions. Our results likely would improve further if a hardware OpenFlow switch was used in the testbed.

Chapter 2

RELATED WORK

2.1 Background on SDN

2.1.1 What is SDN?

In conventional switches and routers forwarding decisions happen based on MAC and IP address, respectively. Both the control and forwarding planes are on the same node in this hardware. What SDN proposes is to decouple these two planes from one another and use a centralized controller [21]. Since all nodes forming the forwarding plane are connected to the controller, the controller has a view of whole network. This makes the controller the best candidate for making forwarding decisions. In addition, the forwarding decision may be based on more fields of the header, even on non-IP based protocols.

2.1.2 Why SDN?

SDN has the potential to simplify network management, and enable innovation in and evolution of computer networks [21]. It is based on the principle of separating the control and data planes. The OpenFlow specification describes the information exchange between the two planes [22]. In this architecture, an OpenFlow switch contains a flow table consisting of flow entries. A flow entry is made up of fields on which incoming packets are matched, and actions to be applied upon a match. If there is no match, the packet is forwarded to a controller, which runs a program to handle the packet, and decide whether to insert, delete, or update flow entries in the flow table for subsequent packets matching the same fields. As well, statistics are collected

on packets; this information may be used by the controller to make decisions. This allows us to build innovative applications which match with our needs, and update them as circumstances change. One such application is load balancing.

2.2 Using SDN in Load Balancing Applications

2.2.1 OpenFlow Based Load Balancing

Similar to our proposed work, Upaal and Brandon [27] investigate whether an OpenFlow based load balancer can compete with existing highly specialized commercial load balancers. Three basic OpenFlow algorithms are implemented and benchmarked. The policies implemented are random, round robin, and load-based load balancing. The random policy sends a request to a random server. The round robin policy uses a circular queue to decide where to send a request. The load-based policy sends a request to the server with the lowest load, where load is defined as the number of pending requests.

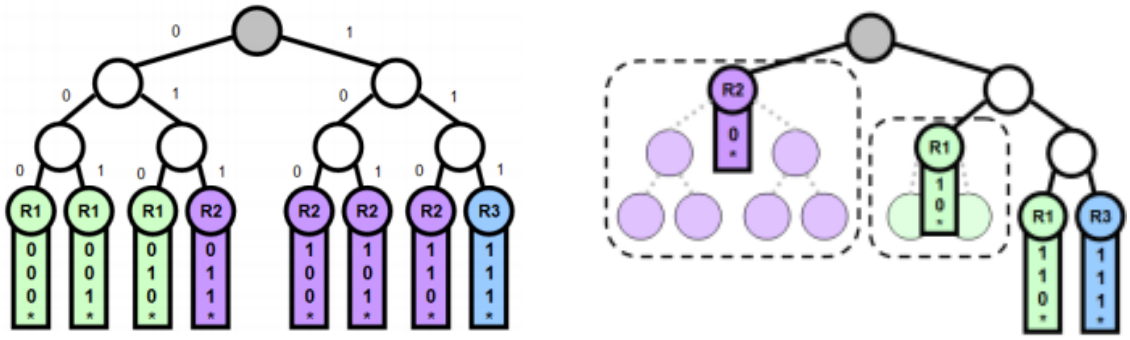
The results indicate that as the processing time per packet at the server is increased, the load-based policy performs the best. When the processing time is 10ms the utilization of the servers is low, so all of the algorithms are essentially the same. Once the time to service of each request increases from 10ms to 20ms, utilization increases. The higher the processing time becomes, the more critical the load balancing algorithm.

In another experiment, two different rules are installed on the OpenFlow switches, one that simply forwards the arriving packets, and another that modifies the packet headers. For the first rule, the performance is as good as an ordinary switch but for the second rule it decreases by two orders of magnitude. The reason is attributed to packets in the buffer being dropped while they are waiting for the OpenFlow switch

to rewrite the packet header. This indicates that preventing bottlenecks in OpenFlow switches is important in order to have a load balancer with a reasonable speed compared with commercial ones. It is expected that in the next generation of OpenFlow switches (or even firmware updates) rule-rewriting will have high performance.

2.2.2 OpenFlow-Based Server Load Balancing Gone Wild

Load balancing in enterprise networks is one potential application of OpenFlow. As Wang et al. discuss, they use a binary tree to represent the space of all possible IP addresses [28]. The ith level in the binary tree corresponds to the ith most significant bits of the IP address. The nodes in a subtree correspond to a prefix match on the path from the root to that subtree. Under the assumption that each IP address supplies equal load on the network, a tree representation is effective since at each level the load is distributed equally between the two subtrees. This solution allows fine granularity of load distribution.



(a) Three resource instances, R1, R2, R3 cover the IP space.

(b) Desirable load balancing configuration with a minimum number of wild card matches.

Figure 2.1: Load Balancing Based on a Binary Tree [28].

Given that ternary content addressable memory (TCAM) is an expensive resource in routers, minimizing TCAM is a useful objective. As Figure 2.1a shows, subtrees

00∗, and 010∗ are assigned to resource R1. The representation in Figure 2.1a is not optimal as it requires six rows in TCAM. There is an equivalent representation that uses less TCAM space as Figure 2.1b shows. Wang et al. provide an algorithm to reduce the tree in Figure 2.1a to the one in Figure 2.1b [28]. In their reduction, they ensure that the connections that are already in place do not get interrupted by the migration. This is guaranteed by making the active connections migrate after they are closed. One weakness of this work is the assumption that load is distributed equally among subtrees. However, the idea of migration is a useful adaptive scheme.

The number of needed wild card matches was reduced by redistributing the IP space among servers. This solution cannot be used in our proposed work because they are trying to slice the IP space among servers while we try to spread the incoming data as equally as possible.

*2.2.3 Aster*x: Load-Balancing as a Network Primitive*

In Plug'n Serve (the precursor to Aster*x), load balancing is studied in an unstructured network [20]. Their interest is in networks that are not built for the purpose of developing server farms such as campus networks and enterprise networks, because background traffic and biased network topologies could affect the performance of network-agnostic load-balancing significantly [20]. The question addressed is whether adding more servers to an unstructured network can improve the overall performance and whether it is possible to devise a general load balancing solution for these types of networks. One aspect to their system design is that all servers have the same IP alias. The controller decides to which server to direct the request. Once the controller chooses a server, it sets up a flow to that server, and the packets are sent at line rate over that flow path.

Aster*x advances Plug'n Serve by performing load balancing on a larger scale net-

work, in particular, over a wide area network (WAN) [18]. In addition, the load balancer can handle the client diversity, serving both local and remote requests. Aster*x has used the Global Environment for Network Innovations (GENI) infrastructure to evaluate its proposed solution [16].

As in Plug'n Serve, in Aster*x all servers use the same IP alias. The controller is in charge of assigning a server to an incoming request. Three different modules are used to make the assignment. Two of them probe network congestion and host load in order to choose the best path, where "best" is the one with the minimum traffic, to the host with the lowest load. The controller then manages the load and routes the flows using the algorithm selected.

The idea of probing the network to understand the load is an important idea. However, our network is not unstructured; as a result there is no significant effect of substantial background load or biased network topologies over the network-agnostic load balancing. We can benefit from their idea of using one alias IP address to refer to a server pool and to put an OpenFlow controller in front of the server pool to act as a proxy and select which server has to receive the load.

2.2.4 Towards an Elastic Distributed SDN Controller

Scalability and reliability are issues that a centralized controller in an SDN enabled network suffers. Having distributed controllers instead of one centralized controller is a solution to these issues. However, a distributed control plane can bring the problem of having the load divided among controllers unevenly. Dixit et al. proposed ElastiCon, an Elastic Distributed Controller, to address this problem [13]. They believe static mapping between a controller and a switch is the main issue [13]. Hence, in their elastic architecture, the controller pool is dynamically grown or shrunk based on the network load and the load could be transferred to another controller with a

reasonable amount of resources when required.

Although migrating switches across controllers is a primitive in their design, it is not sufficient. Additional mechanisms are provided to support three main operations to balance the load across the controllers. First, there is a mechanism in charge of balancing the load among controllers by reassigning switches to controllers with sufficient resources periodically. Secondly, there is an upper margin which indicates the total load that could be handled by the resource pool; if the load exceeds this margin the resource pool has to be grown. Finally, there is a lower margin which detects when we need to shrink the resource pool.

Although the idea of balancing the load among controllers by measuring their utilization level is a very good way to make switching and or routing decisions faster, it is out of the scope of our work.

2.3 Testbed

In this section we introduce some of the software and tools that we use in our research.

2.3.1 XenServer

XenServer is a free product from Citrix that is used for server virtualization [11]. This open-source virtualization platform has been used in cloud services as well as server and desktop virtualization. In our project XenServer helps us to install and manage all the needed virtual machines (VMs) on the physical machine. It is responsible to allocate resources to the VMs.

2.3.2 *Mininet*

Mininet is an open-source network emulator tool [19] [2]. Using this tool we can imitate any arbitrary network topology with all of its components. It provides end-hosts, switches, links and routers on a single Linux kernel. Using lightweight virtualization it makes a single system look like a complex network. We can run any software that is compatible with Linux, from web servers to Wireshark on its end-hosts. Mininet also gives us the ability to customize our packet forwarding using the OpenFlow protocol. It has the ability to establish a connection between the emulated network to a local or remote OpenFlow controller to obtain the packet forwarding decisions. In this research this useful tool helps us to test the controller before running it on the testbed.

2.3.3 *Open vSwitch*

Open vSwitch (OVS) is a distributed software switch [3]. While this software supports standard management interfaces and networking protocols, its main goal is to provide switching for hardware virtualizing platforms in a multilayer virtual switch. This software allows network automation via its programmatic extensions. It had been used by XenServer and Xen Cloud Platform as their default switch [3]. In this research we use this software as our OpenFlow enabled switch.

2.4 Controllers

There are several popular OpenFlow controllers written in different languages offering a wide variety of services. Some of these controllers include:

POX: POX is a good choice for rapid development and for developers who want to prototype a network controller software. This controller is written in Python.

Its high-level SDN API allows developers to quickly turn their ideas into reality and helps them to write their own OpenFlow controller.

OpenMUL: OpenMUL is an SDN/OpenFlow controller whose core is written in C [12]. Its main goal is to reach a high performance and reliability by virtue of its multi-threaded core. It has a graphical user interface (GUI) and a web service API that is bound to representational state transfer architectural limitations (RESTful API) [14] alongside the command line interface (CLI) that makes the process of its management much easier. The RESTful web services are used to return JavaScript Object Notation (JSON) and XML format in response to application-specific web URLs. JSON is a lightweight human-readable open standard format for data transmission. It was first used as an alternative to XML to send data between a server and a web application. This controller supports a variety of SDN south-bound protocols such as OpenFlow 1.0, 1.3 and 1.4 as well as ovsdb and of-config.

Floodlight: Floodlight, an Apache-licensed OpenFlow controller, is a Java-based controller that can be used for controller development in enterprise networks [1]. This controller is a derivative of another Java-based controller named Beacon [1]. Floodlight has been used in a commercial product from Big Switch Networks as core of their network controller software [7].

FlowVisor: FlowVisor, a special purpose OpenFlow controller, is designed to support network virtualization [24]. FlowVisor has the ability to make slices of network resources and assign each one of them to a different controller. Network resources can be sliced at any layer, *e.g.*, they could be sliced at layer one in terms of any combination of switch ports, or they could be split at layer two in terms of source/destination ethernet address or type; they also could

be distinguished in terms of source/destination IP address in layer three or as source/destination UDP/TCP ports in layer four.

OpenDaylight: Big companies such as Cisco, Microsoft, Citrix, Juniper, Intel and IBM contributed to develop the OpenDaylight SDN/OpenFlow controller [26]. In this project these industry leaders come together with one goal, and that is to provide a common SDN platform to let other businesses and companies use it as their base SDN solution. Based on each company's needs, developers can utilize the code or even develop new features that can fit into their requirements.

2.5 Splunk and Syslog

Splunk is software that is designed to let its users search, monitor and analyze machine-generated big data through its web interface [6]. Machine-generated data includes all data logs that are produced by applications, servers, network devices, firewalls, *etc.* One of these log data can be syslog [15] data coming from network devices such as routers and switches. This syslog data can record the network's status including device failures, security threats, performance, and the state of network connections. Syslog has being used as a common computer message logging standard for years. Rsyslog is software that is used to store syslog messages on a server. At the University Technology Office (UTO) of Arizona State University (ASU) Splunk is being used to capture the log data coming from the Palo Alto (PA) firewalls into a web-style format that allows network administrators to monitor and analyze these logs more easily.

Filtering is one of the features that Splunk provides for its users. It allows users to prevent some of the logs from being indexed in Splunk database. For instance, a user can define a rule to filter all the logs that are initiated at a specific machine by its IP address.

2.6 Current Solution

UTO uses Citrix NetScaler [9] as their current load balancing solution in front of Rsyslog servers. NetScaler divides the IP address space and spreads the load among servers based on source IP address of incoming packets. As a result, NetScaler divides the whole load coming from one PA firewall to one Rsyslog server. Therefore only one of the Rsyslog servers receives the whole load and the rest of them receive nothing. With multiple PA firewalls this leads to a very uneven load distribution among Rsyslog servers.

In the next chapter, we discuss the testbed setup, and our controller design to balance on load differently, rather than on the IP address space.

Chapter 3

ARCHITECTURE OF OUR SDN LOAD-BALANCING SOLUTION

In this chapter we discuss the architecture of our OpenFlow based load balancing controller design and how we use it to improve the current solution for processing the data coming from Palo Alto (PA) firewalls.

The main concern with our proposed architecture is whether it is fast enough to keep up with the data rate. We describe the configuration of the testbed, and how we verify its connections. We conduct experiments in the physical testbed, gather data, and analyze our results in Chapter 4.

3.1 The Testbed

3.1.1 Topology and Technical Specification

We are given a wire tap to the data that one of the PA firewalls is sending out. This wire tap is directed at our testbed to feed the load balancer for examination and verification purposes. (Permission to obtain this tap was granted by the Research Administration Office of the Arizona Board of Regents for and on behalf of ASU.)

Our testbed includes a physical Dell server with 32 GB of RAM and 32 Intel Xeon E5-2640 v2 processors, each one working at the speed of 2.00 GHz. On the Dell server we installed Citrix XenServer version 6.2. Then we installed four virtual machines (VMs) on the XenServer. We allocated 4 GB of RAM to the first VM and 8 GB of RAM to each of the other three. Each VM is allocated 4 Xeon CPUs.

In an OpenFlow network there is a central controller that is responsible for making routing and switching decisions. The first VM is configured to act as our controller

and the other three are set up as Rsyslog servers. As Figure 3.1 shows, the Open vSwitch (OVS) is installed on the XenServer. Our software switch is connected to the controller and the three other VMs. The OVS is configured to have the OpenFlow protocol enabled on its bridge xenbr0. As a result it sends every new flow to the controller and then the controller decides whether to install a rule on OVS for similar flows or handle them on a packet by packet basis.

3.1.2 The Data Feed and its Features

The data coming in to the testbed is the data sent out by the PA firewall. The data is in syslog format. The existing PA firewall at ASU uses the UDP protocol to send out its syslog messages. While more recent PA firewalls give the option to send these syslog messages in TCP packets, we restrict our attention here to UDP.

Recall that the UDP protocol is not a reliable transport protocol. As a result, the syslog messages might get lost and there is no mechanism to retransmit those packets. We cannot guarantee there are no packet losses. However our objective is to deliver as many packets as the current solution delivers to the destination. In Chapter 4 we design an experiment to perform this validation.

Rsyslog messages are smaller than the maximum transmission unit (MTU) of IPv4 packets. As a result these packets do not get fragmented. Hence, in our design there is no need to take fragmentation into consideration.

3.1.3 Setting Up the Testbed

Since we use OVS to imitate the behavior of an OpenFlow switch, we must have OVS running on the XenServer. In our case there are four different network interfaces on the physical machine. eth0, ..., eth3. The XenServer has the ability to manage them through the OVS. In order to do so, when XenServer is booting up it creates four

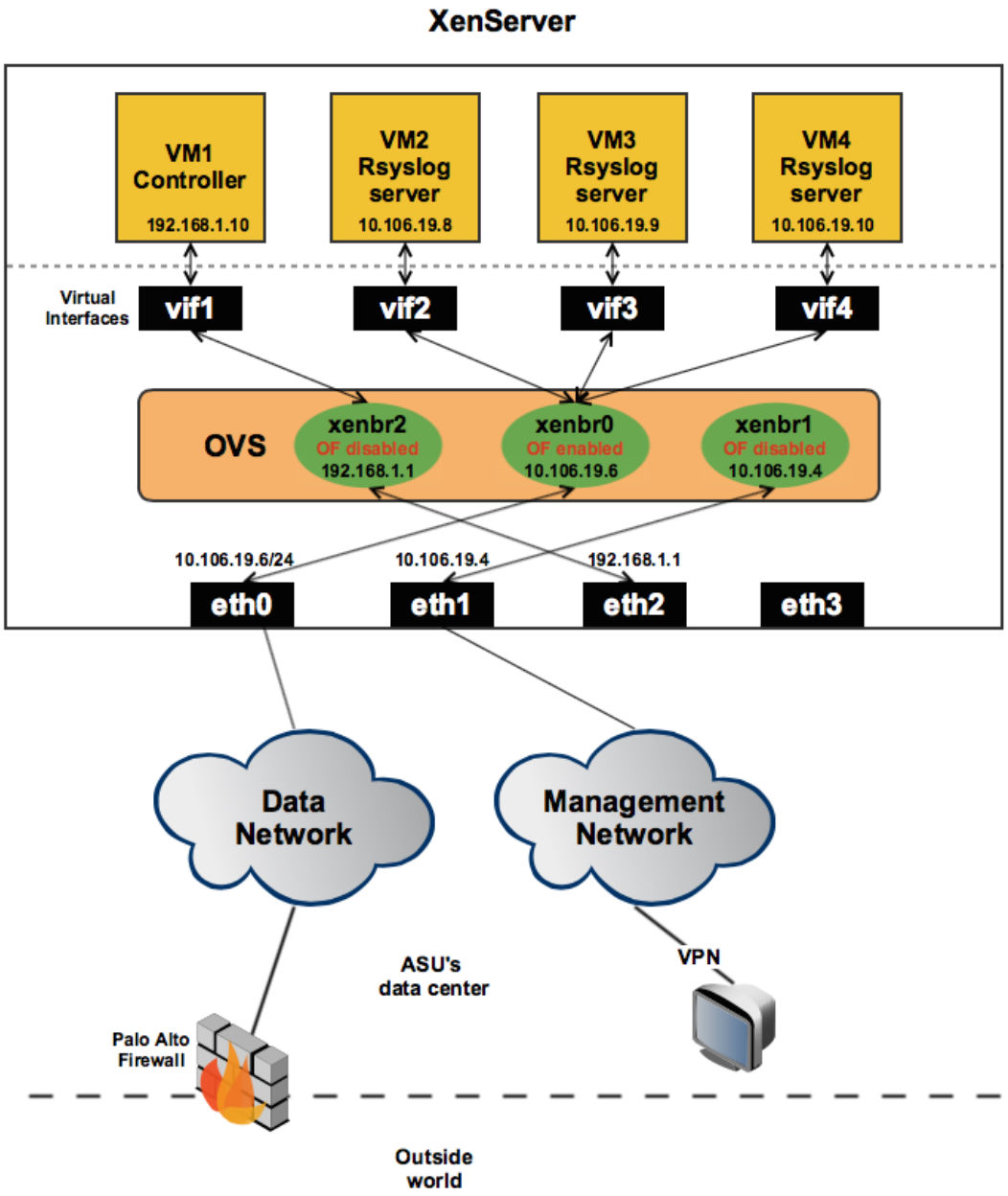


Figure 3.1: The Testbed Topology.

different bridges xenbr0, ..., xenbr3, using OVS and creates a one-to-one mapping between each interface and bridge, i.e., eth0 is added to xenbr0 as a port.

When we want to change any IP settings on a physical interface, we have to make the changes on its mapped bridge interface. From now on, whenever we refer to configuring physical interfaces such as eth0 and eth1, we are referring to configuring their mapped bridges. We use all of these bridges to set up our network. Recall Xenserver is only accessible over the network through its eth0 and eth1 interfaces.

Since the PA is on the local campus network, our testbed must be connected to that network to obtain the data coming from the PA. The PA forwards its packets to an IP address in the range of the network addresses we assigned to our testbed. In our case the network address that the testbed is using is 10.106.19.0/24 and the PA forwards its data to the IP address of 10.106.19.7. We have set up eth0 to use an IP address of 10.106.19.6. In this case eth0 is a hop in the data path from the PA to the Rsyslog server. We call this network, 10.106.19.0/24, our data network.

We also need to have access to the XenServer through another interface to reach the XenServer remotely to set up the VMs, configure their attributes, and control them. We call this network our control network. There are three reasons we cannot use our data network to connect remotely to the XenServer for control purposes:

1. OVS is using our controller to make its forwarding decisions on xenbr0.

2. eth0 is on xenbr0.

3. Our controller only forwards those packets that are targeting to reach to the Rsyslog server.

As a result any control or management packets on xenbr0 destined to go back to the computer that is remotely connected to the Xenserver, are dropped by the OVS.

It is ideal having the management network and the data network to use two different network addresses, as well as gateways, to prevent any conflict between them. When we make any changes on one of the networks it does not affect the other one, since they are completely separated. In our situation due to some restrictions from the University Technology Office, we use the same network address for both networks. We assign the eth1 interface an IP address of 10.106.19.4. Recall that this interface is on xenbr1 which is not controlled by our controller and is a non-OpenFlow bridge. Hence there should not be any problem with connecting to this interface remotely.

Since eth0 and eth1 are both on the same network we have to make sure when we are remotely connected to eth1 all of our management packets are going through this interface. In order to test connectivity, we unplug the cable connected to eth0. Now when we want to SSH to 10.106.19.4 from a remote machine we get an error indicating the server does not exist. The problem is that in our case we have set up eth0 first and then eth1. This leads to having two different routes to the same destination in the routing table on the XenServer. The routing table is shown in Figure 3.2. In this case when management packets want to reach to the remote machine they match the first match in the routing table which is eth0. To solve this problem we need to reverse the interface configuration order. We can also solve this issue by disabling and re-enabling the eth0. This changes the order of entries in the routing table and leads to matching the correct entry while we are trying to reach to the XenServer remotely.

Now that we have set up our data and management network we should set up our Rsyslog servers. We install Rsyslog on the VMs and start this software running so we can process syslog messages coming from the PA. We install Rsyslog on VM2, VM3 and VM4. We have to set up these VMs in such a way they can interact using our

```
[root@xenserver-redefine ~]# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.106.19.0     *               255.255.255.0   U     0      0        0 xenbr0
10.106.19.0     *               255.255.255.0   U     0      0        0 xenbr1
default         10.106.19.1     0.0.0.0         UG    0      0        0 xenbr1
```

Figure 3.2: Erroneous Routing Table on Xenserver

controller. We assign an arbitrary IP address in range of 10.106.19.0/24 to each VM. Then on the XenServer we add their interfaces to xenbr0 on OVS.

To finish configuring our OpenFlow enabled network, we need to configure xenbr0 on the OVS to connect it to the controller. Our controller is running on VM1. In OpenFlow, the controller has to communicate with a switch on a non-OpenFlow network. In order to accomplish that we configure the VM1 networking interface to have a different network address than the data network. We assigned 192.168.1.10 to the VM1 interface. We need to configure the XenServer to communicate with VM1. To make this happen we first assign an IP address in the same range to one of the free interfaces on XenServer. We use eth2 and set it up to work with the IP address 192.168.1.1. VM1 interface needs to be added to the same bridge that eth2 is on. We add the VM1 interface to xenbr2. The last step is to inform xenbr0 where to find its controller. We set up that using the following command:

ovs-vsctl set-controller <bridge> tcp:<ip>:<port>

To do all the configuration of VMs we use software provided by Citrix, called XenCenter [10]. This software connects to the XenServer remotely and gives full control over the physical machine and VMs. There is also a command line interface (CLI) that allows control of VMs and their attributes. But XenCenter makes it very convenient to control everything through the GUI it provides.

This software provides us a console to each VM. Through this console we can access each VM and configure the required attributes. In the validation phase of

this project we use the same console to reach each one of the VMs and perform the verification. Using XenCenter on a VM, higher level configuration such as adding a virtual network interface to a VM or assigning that interface to XenCenter's bridges as one of their ports, is also supported.

To complete the testbed setup, we need to create another OpenFlow disabled network including all the VMs. As shown in Figure 3.3 this network is for probing purposes. We will see that in the load-based policy, we use this network to probe all of the Rsyslog servers. By probing Rsyslog servers at the time of server selection in the load-balancing process, we can prevent the controller from forwarding data to a server that is down. In order to add this network, we have to add a second interface to each one of the Rsyslog servers and a third network interface to the controller. Recall that second interface on the controller is dedicated to a TCP channel between the controller and the OVS. After adding another interface to all of the VMs we have to put them all on the same OVS bridge. We assign all of them to xenbr3 on OVS. Each VM has been assigned an IP address in range of 192.168.2.0/24.

3.2 Controller Design

In this section we explain different components in the controller and how they work together. As we know, the controller is in charge of making all the forwarding decisions in our data network. Our goal is to divide the load among the Rsyslog servers in such a way that one Rsyslog server receives the load at a time. We develop three policies in which the objective is to spread the load in such a way to make these files as even in size as possible.

1. Round-Robin: In this policy we assign each server a static ID starting from zero, and incrementing by one. For the first time around, we choose ID number zero, and store the ID number of the selected server in a static variable. Every

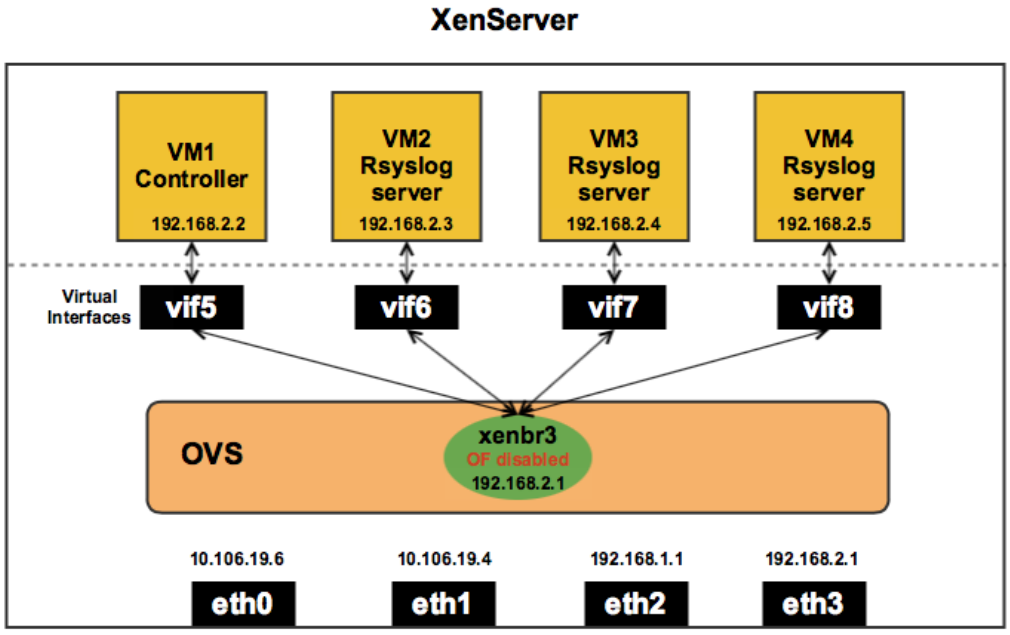


Figure 3.3: OpenFlow Disabled Network Setup for Network Probing.

time this method is called, we retrieve the ID number of last selected server, increment it by one and perform a modulo n operation on it, where n is the total number of Rsyslog servers.

2. Random: In this policy we choose our next server at random.

3. Load-Based: In this policy we choose the server which has received the least amount of data for today.

Since the first two policies are the most common policies in load balancing, we have chosen to implement them as well. The reason for implementing the Load-Based policy is that we believe the first two policies cannot operate well in a network where server failures occur.

Before describing different components in the controller, we explain how OpenDaylight works and how we should develop a controller using OpenDaylight. Open-

Daylight takes advantage of the following tools and software:

- Bundle: A bundle consists of a group [5] of Java classes that form an application and come with a manifest file containing information about these classes, as well as required packages and services that are needed by these groups of Java classes.

- OSGi: The Open Gateway Service initiative [4] is a set of specifications that enables system modularity. It installs bundles and allows them to exchange their information using a service model without revealing their content. It also allows to remotely install, update, uninstall, stop or start, a bundle.

- Maven: Maven is a project management tool [25]. It uses a project object model (POM) xml file to detect dependencies for the project, needed packages to download, where to look for those packages and what bundles to start. OpenDaylight uses this software to automate the process of building a project.

In OpenDaylight, our controller is nothing but a bundle that uses other bundles' services to provide a load-balancing application for a specific use. Next we explain in detail the different components in our bundle.

3.2.1 Initialization

Prior to initializing our bundle, we need to uninstall three of the bundles that are running by default in order to prevent any interference with our bundle. These three bundles are:

- ARP Handler: As its name suggests this bundle is responsible for handling all of the ARP packets.

- Simple Forwarding: Simple Forwarding is an application that makes simple forwarding decisions like a conventional switch, and installs rules related to the

forwarding decisions across the entire OpenFlow network. It has the ability to discover hosts using ARP messages.

- Load Balancer Service: This built-in load balancer application installs OpenFlow rules reactively. It can be configured through the REST API to forward all packets with a specific source and port address to one its backend servers. Random and Round-Robin are two policies that this bundle includes.

We stop these three bundles because they interfere with the logic of our bundle. When the ARP Handler is running it responds to ARP requests and it does not allow our bundle to receive those packets. Therefore, if we do not stop the ARP Handler our bundle cannot create a mapping between MAC addresses and IP addresses using ARP messages. Although not stopping the ARP Handler does not mean that our bundle would not work properly, it is better to stop this bundle so we can handle ARP messages through our bundle and use them to create a mapping between IP and MAC addresses faster.

The Simple Forwarding bundle also does not match with the logic of our bundle. It simply finds routes to hosts and writes rules for those flows. But our requirement is to change the packet destination for load balancing purposes periodically. As a result this bundle interferes with our application and is therefore stopped.

The last bundle we stop before starting our bundle is the default load balancing bundle. By running some experiments we determined that this load balancer cannot divide the UDP load among our Rsyslog servers. Although this bundle needs to be configured through the REST API to start working, we prefer to prevent any unpredictable behavior this bundle may cause by stopping it.

The PA firewall directs everything to one IP address. In our testbed, that IP address is an alias IP address, meaning that there is no Rsyslog server running with

that IP address. Now it is our job to redirect everything that the PA sends out, to real Rsyslog servers. The IP addresses of the Rsyslog servers and the IP address of an alias Rsyslog server are given to our bundle via a file on the controller. We have to know these IP addresses for further processing and packet manipulation. In this file a 32-bit integer value is provided as well. This value is the hard time-out for OpenFlow rules our bundle writes on the OpenFlow switch. The configuration file format is:

Policy = *Policy name*
Hard timeout = *m seconds*
The repeating the same selection limit = *t times*
Alias IP = *IP address of alias Rsyslog*
Server IP = *first Rsyslog server IP address*
Server IP = *second Rsyslog server IP address*
.
.
.
Server IP = n^{th} *Rsyslog server IP address*

We will explain the repeating the same selection limit in the first step of the *Handling UDP Packets*, subsection 3.2.3.

3.2.2 Processing Incoming Packets

When a packet comes into the OpenFlow enabled switch, the switch tries to find a match for the packet in its forwarding table. If the switch fails to find a match for the packet, it forwards the packet to the controller. OpenDaylight receives the packet and it notifies all the bundles that have implemented its *IListenDataPacket*

interface. To process an incoming packet in a bundle, the bundle has to override the *receiveDataPacket* method in the *IListenDataPacket* interface. This method returns a type known as *PacketResult*. *PacketResult* can acquire three different values.

1. *PacketResult.CONSUME*
2. *PacketResult.IGNORED*
3. *PacketResult.KEEP_PROCESSING*

The *CONSUME* value notifies OpenDaylight that our bundle has taken care of this packet and no other bundle in the chain after us should get a copy of this packet. The *IGNORED* value informs OpenDaylight our bundle has not processed this packet. Hence, a copy of this packet has to be forwarded to other bundles for further processing. The *KEEP_PROCESSING* value indicates that the packet has been processed by our bundle and further processing is still possible by other bundles. As a result the controller forwards a copy of the packet to other bundles.

Now that we know how to get a copy of incoming packets from OpenDaylight, we must describe how to process them. When our bundle starts working, it does not know anything about the network's topology or the servers' MAC addresses. So when we get a new packet we extract this information into two different hash tables. We are interested in knowing what IP address corresponds to what MAC address, and what MAC address corresponds to which port on the switch. Without knowing this information we cannot write proper OpenFlow rules on the switch. Hence we need to collect this information before we can start writing any rules.

Since the PA encapsulates its data in UDP packets, after extracting the IP and MAC information we check to see whether the packet is a UDP packet. If it is a TCP packet we ignore it and leave it to other bundles to process. If it is anything other

than UDP we inform the switch to flood these packets, depending on what mode the bundle is operating, but we do not write any rule for them. Our bundle operates in two different modes. The first mode is an initialization mode, and the second mode is a post-initialization mode. Initialization mode indicates that the bundle has not collected all the information required for writing OpenFlow rules. As a result it keeps broadcasting all packets except TCP and UDP packets. When the bundle collects needed information, it transitions to its post-initialization mode. This means the controller stops broadcasting any packets except ARP messages. Next we go into detail of handling UDP packets.

3.2.3 Handling UDP Packets

When we process UDP packets we are only interested in packets destined for our alias Rsyslog IP address. Therefore, we do not further process a UDP packet if it is not intended to go to an Rsyslog server. To route a UDP packet with syslog content we perform the following steps:

1. *Use the load balancing policy to select a server.*
2. *Create a match for packets similar to the current packet.*
3. *Make a list of actions to be made on any match with this packet.*
4. *Write the rule for this flow in the switch.*
5. *Forward the current packet.*
6. *Periodically, proactively update this flow to forward packets to the next server according to the load balancing policy.*

Selecting a Server

We select a server from the server pool based on the policy we are using to spread the load. We assign an integer to each server starting from zero and incrementing by one. We call this number our *server ID.* We put these values in a hash table. This makes selecting a server from an existing server pool and tracking previous choices easier. Then based on the policy we are using, we select a server as the destination of packets.

To make sure we are not sending the packets to a server that is down or to a server that is having trouble listening on port number 514 (this is the default port to receive syslog messages), using a script we first probe port 514 on all of the servers to determine which of them is up and listening. All the syslog servers listen on port 514 for TCP and UDP packets. Since there is no way to verify delivery of a probing packet using UDP, we use TCP to probe port 514 on the servers. There are two different methods to probe a TCP port. One is to connect to that TCP port, i.e., open a connection and tear it down immediately. The other is to only send the SYN packet to the TCP port and wait for a SYN-ACK packet. If the server sends the SYN-ACK packet it means it is up and listening on that port. We use the second method, since it is slightly faster and does not require the Rsyslog daemon on the server to create another process to listen on the same port. To implement second method we use Nmap [17], software that allows us to scan ports on a computer network, and provides us a result indicating whether specific port is open on each host. At first we wanted to use this software to probe the network, but for unknown reasons the probing process was very time consuming. For instance instead of probing a server in a fraction of a second, it took about twenty seconds for Nmap to produce a report. As a result we decided to use the setup connection method instead.

The code to probe the servers is written in Python. This code is called at the beginning of the server selection process. It writes the probing results in a file. Later in our program, the bundle reads the results from the file and uses them to check which servers are available in the server pool.

Different load balancing policies use this information about available servers differently. The Round-Robin policy first selects the next server simply by incrementing the server ID from the last time. Then it checks to see if that server is up and running. If it is not up, increments the server ID and checks again. This process continues until the selected server's status is up.

The Random policy first uses the information in the file to create an array of server IDs whose status is up, then it selects a number in range of the array's length at random; this number is the index of selected server ID in the array.

The Load-Based policy chooses a server based on the number of bytes the switch has sent out on each port. An OpenFlow switch has the ability to keep track of statistics such as the number of packets it has received and sent out on each port, the number of errors in receiving and transmitting packets, the number of incoming and outgoing dropped packets, *etc.* Since OpenDaylight is a RESTful controller, to obtain such statistics we query the controller by sending a specific web URL to the controller on port 8080. The OpenDaylight controller has a web server running on that port by default. When we send out the query to the controller to obtain the statistics, the controller processes it and sends the corresponding command to the switch to get the information. The statistics that come back from the switch are in JSON format. Our bundle parses the JSON report to obtain the number of bytes that is sent out on each port. Since the switch keeps this information for each port, we have to use a hash map to figure out what servers are on what ports.

Rsyslog can separate incoming logs based on the server's local time. For instance

it can put all the logs in a file on an hourly or a daily basis. In our case Rsyslog splits the logs on a day to day basis. Our objective is to make the files for each day as equal in size as possible. Therefore we need to derive number of bytes we have sent out for the day from the total number of bytes that we have sent out from the beginning of controller's job. As a result we need to keep track of how many bytes we have sent out on each port starting from midnight of each day. Recall only those ports that go to a server are important to us. By subtracting the number of bytes we sent out at midnight from the number of bytes we have sent out up to this moment, we obtain the number of bytes we have sent out to the current time. Using our hash map we can detect which server corresponds to the fewest number of bytes that we have sent out on all ports. In this calculation there is one thing we need to be cautious about, and that is overflow. As the OpenFlow specification declares, the number of received bytes is stored in a 64-bit unsigned integer. Hence if we get a negative number in our calculation, an overflow has occurred. To take care of this overflow, we need to add up the following values to get the number of received bytes so far for the day for that particular server:

- *The difference of the maximum positive value an unsigned 64-bit integer can store and the number of received bytes at midnight*
- *The number of received bytes so far for the day*

As we will see, the different policies behave differently when a server that has been down for a while comes back up. The Round-Robin and Random policies do not try to compensate for the time the server was unreachable. They continue their normal routine. They only use the server that just came back up to split the load. On the other hand, the Load-Based policy keeps selecting the server that just came back up to compensate for the time it was down. There is an option in this policy

that can be set to put a limit for selecting same server repeatedly. This option could be set in the configuration file in integer format. We call this number the repeating the same selection limit (RSSL). The RSSL determines how many times a server can be selected consecutively. This option not only can prevent other servers from being idle, but also can help to keep the server that was down from being overwhelmed with data. The smaller the RSSL, the longer the compensation time lasts, where compensation time is the time it takes for controller to fill the created gap between the server that was down and the rest of the servers.

Creating a Match

In the second step we have to create a matching rule. We match a packet based on following fields:

- *EtherType*
- *Network Protocol*
- *Network Destination Address*
- *Destination Port Address*

EtherType is a two-byte field that shows which protocol is encapsulated inside an Ethernet frame. For those packets going to the Rsyslog server this value is 0x0800 which indicates that the Ethernet frame contains an IPv4 payload.

The *Network Protocol* is a one byte field indicating the protocol encapsulated in the IPv4 frame. If this field is equal to 17, the IPv4 packet encapsulates a UDP packet.

The *Network Destination Address* is the destination IP address which is represented in integer type. The packets that we are interested in forwarding have a

destination IP address set to the alias Rsyslog server.

The *Destination Port Address* is represented in a short type, indicating the destination port where packets are destined. By default, Rsyslog listens on port 514.

Since our purpose is to forward all the traffic to one server at a time and there are multiple PA firewalls in principle, we cannot include the source IP address in the fields to match. For the same reason we cannot include the source port address.

Making a List of Actions

We now make a list of actions to be taken on the packets that have the same fields as our match. These actions are:

- Rewrite the destination MAC address of the packet with the destination MAC address of the selected server.
- Rewrite the destination IP address of the packet with the destination IP address of the selected server.
- Forward this packet on the outgoing port that reaches the selected server.

Writing the Rule

In the fourth step we write a flow on the switch. This flow includes a match and a set of actions to be taken on that match. To write a flow on the switch we need to specify a hard and soft timeout for that flow. A hard timeout defines the expiration time of the flow after the flow is installed. A soft timeout specifies to remove the flow after a flow is idle for the given period of time. The hard timeout could be set by the user in a configuration file placed on the controller. We do not set the idle timeout, since the PA never stops flooding. As a result before this flow becomes idle we need to rewrite another flow to forward the load to another server in order to prevent losing packets.

Forwarding the Current Packet

In this step we have to forward the current packet to the selected server. Otherwise we lose those packets that do not match the rule on the switch and are forwarded to the controller for further processing. To do so we have to rewrite some parts of the IP packet and UDP packet. The checksum in UDP packet needs to be updated to reflect the IP address of the selected server. The UDP checksum is a one's complement sum over the IP header, UDP header and data [23]. For the new packet, the UDP header and the data remain the same. But the IP header changes. Hence we have to update the UDP checksum reflect the change. To update the checksum we use the following rule:

New checksum = Old checksum - (IP address of the alias Rsyslog server - IP address of selected server)

We also have to rewrite the IP address and MAC address parts in the IP packet. Then the packet is ready to be sent out on the proper outgoing port.

Updating the Current Flow

The last step is to update the current flow periodically. Since the speed of the data coming in is fast, on the order of hundreds of megabits per second, we need to write the flow proactively. If we write the rule reactively we might lose some of the packets due to the high incoming data rate. The period of updating the flow is relative to the hard timeout for the flow. We update the flow α seconds prior to its expiration time. α is defined as the time it takes for the controller to do all required actions for updating the flow table and rewrite the existing rule. By doing this we would not allow the switch to remain without a proper rule for any incoming flow of interest.

In the next chapter we design two experiments to test our controller and compare

results that we obtain from different policies, and to the existing commercial load-balancer.

Chapter 4

EVALUATION OF SDN LOAD-BALANCING POLICIES

In this chapter we design two experiments, one is to verify that our solution is reliable and can deliver the data that the current solution delivers. The other verifies that our solution splits the load among the servers as the current solution. We also compare the three SDN load-balancing policies to see which one performs better in terms of balancing the load.

4.1 Design of the Experiment

The Palo Alto (PA) sends its log data to ASU's Rsyslog servers as well as to our testbed. Therefore the data we are receiving is a replica of what is being sent to ASU's Rsyslog servers. However, the path to our testbed is different from path going to ASU's servers and the data is being sent in UDP. As a result there is no guarantee that we get exactly the same data as ASU's Rsyslog servers. As we will see, this is why sometimes we deliver more data than the current solution.

We set-up and run two experiments. In the first experiment we run each policy for at least five days and we gather the results. During this period none of our servers experienced any failures. In the second experiment we simulate the condition of losing one or even two of the servers for some time, by taking the servers down, to see how each policy handles server loss. We explore how well each policy operated in terms of data delivery and splitting the data as equally as possible among Rsyslog servers.

4.2 Data Delivery Verification

The best method to check the data delivery is to feed the log files collected by our Rsyslog servers on the testbed to Splunk and see if we are getting the same results as the direct feed to ASU's Rsyslog servers. Since Splunk licensing is based on the amount of data we feed Splunk each day, and the fact these log files are huge (on the order of tens of giga bytes), UTO decided not to use this verification method.

Another way to verify the data delivery is to compare log files that we gather with the log files that UTO stores in ASU's data centers before feeding them to Splunk. Since these log files are huge and we cannot enter the data centers because of security reasons, this verification method is not feasible either.

One other way that we can compare our results to what UTO obtains is to compare the number of syslog messages that we recieve to the number of syslog messages that Splunk indexes in its database. As it is described in syslog standard [15], each syslog message has to be sent in one line. As a result, each line in the log files that we store on our servers is a syslog message. We can count the number of lines in each file and obtain the number of syslog messages in a file using a Linux command. However, we have to consider Splunk filtering. As these logs are generated in ASU's production network, UTO filters the logs for management purposes. Hence, we need to perform the same filtering as UTO before comparing our numbers. UTO filters some of the logs based on IP addresses. We use a Linux command to count the number of lines in the log files except those lines that contain certain IP addresses. After that we add up the numbers that we get from each log file and compare the sum with the number that Splunk gives us. The delivery ratio is defined by dividing the sum by the Splunk's number. We present results for each policy.

4.2.1 Round-Robin Policy

In the first experiment, there is no server failure. As can be seen in Figure 4.1 and Table 4.1 the number of syslog messages that we obtain after filtering is very close to what Splunk indexes into its database.

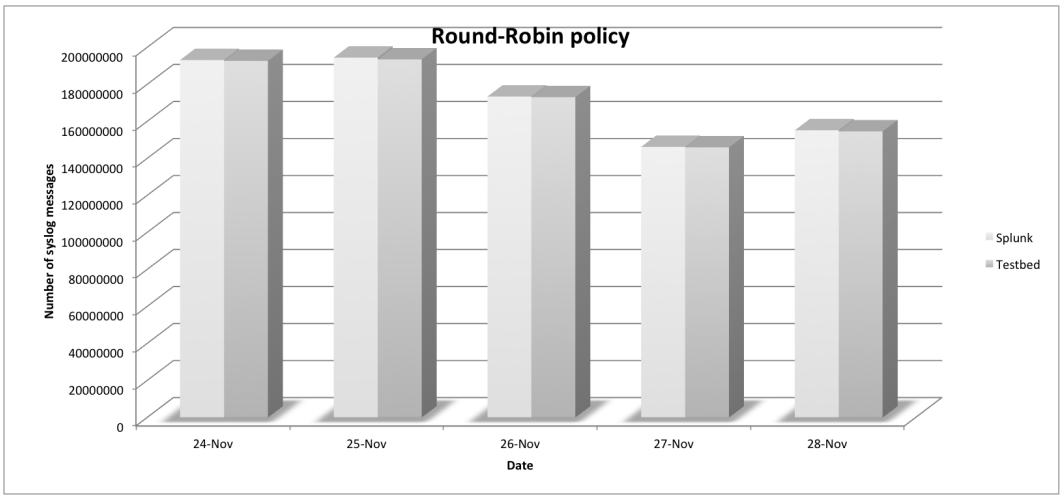


Figure 4.1: The Number of Syslog Messages After Filtering on the Testbed Compared to the Number of Messages Splunk Indexed Into Its Database for the Round-Robin Policy With No Server Failures.

In the second experiment we simulate server failure by shutting servers down. As Figure 4.2 shows, on December 19th we experienced an unusual gap between what we stored on our servers and what the Splunk report shows. Numerical results for this experiment are in Table 4.2. But the average delivery ratio (number of syslog messages we stored on testbed divided by number of syslog messages Splunk indexed for each day) over this period was still 0.993.

Date	Testbed (T)	Splunk (S)	Difference ($S-T$)	Delivery ratio (S/T)
Nov 24$^{\text{th}}$	192664117	193157905	493788	0.997443604
Nov 25$^{\text{th}}$	193478076	194470616	992540	0.994896196
Nov 26$^{\text{th}}$	173037862	173405381	367519	0.997880579
Nov 27$^{\text{th}}$	145913198	146199972	286774	0.998038481
Nov 28$^{\text{th}}$	154588059	155168922	580863	0.996256576

Table 4.1: Total Number of Syslog Messages on the Testbed After Filtering Compared to the Number of Messages in Splunk Report for the Round-Robin Policy With No Server Failures.

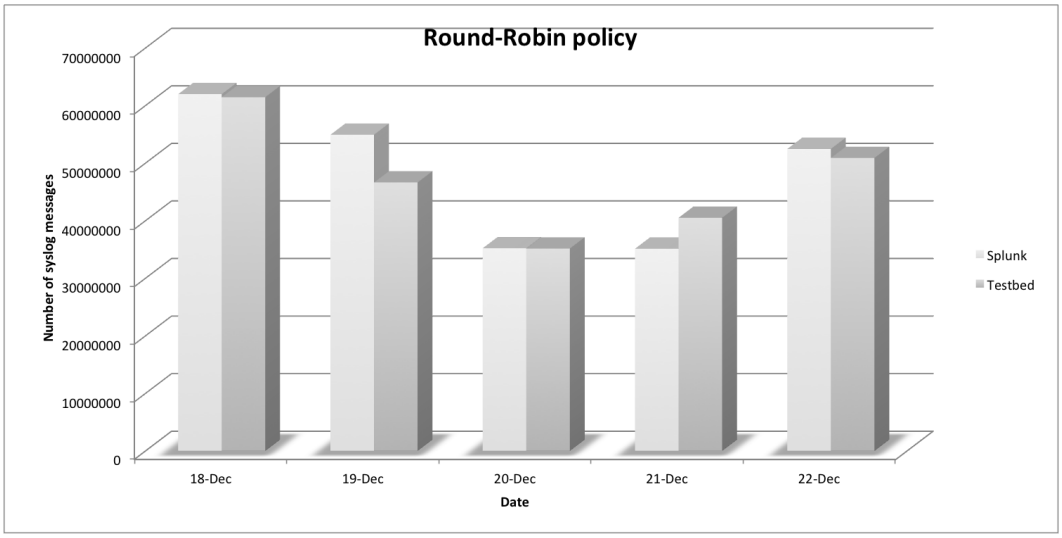


Figure 4.2: The Number of Syslog Messages After Filtering on the Testbed Compared to the Number of Messages Splunk Indexed Into Its Database for the Round-Robin Policy With Server Failures.

Date	Testbed (T)	Splunk (S)	Difference ($S-T$)	Delivery ratio (S/T)
Dec 18$^{\text{th}}$	61450166	61974048	523882	0.991546752
Dec 19$^{\text{th}}$	46629441	54922906	8293465	0.848998066
Dec 20$^{\text{th}}$	35155013	35214334	59321	0.99831543
Dec 21$^{\text{th}}$	40500742	35132478	-5368264	1.152800608
Dec 22$^{\text{th}}$	50886551	52465403	1578852	0.969906797

Table 4.2: Total Number of Syslog Messages on the Testbed After Filtering Compared to the Number of Messages in Splunk Report for the Round-Robin Policy With Server Failures.

Date	Testbed (T)	Splunk (S)	Difference ($S-T$)	Delivery ratio (S/T)
Dec 1$^{\text{st}}$	216021445	216497873	476428	0.997799387
Dec 2$^{\text{nd}}$	220307032	221506328	1199296	0.994585726
Dec 3$^{\text{rd}}$	213855086	214371744	516658	0.997589897
Dec 4$^{\text{th}}$	217260686	218149382	888696	0.995926204
Dec 5$^{\text{th}}$	189661407	189555048	-106359	1.000561098

Table 4.3: Total Number of Syslog Messages on the Testbed After Filtering Compared to the Number of Messages in Splunk Report for the Random Policy With No Server Failures.

4.2.2 Random Policy

As Figure 4.3 depicts, the Random policy performance was very good in terms of data delivery for the first experiment with the average delivery ratio of 0.997. These results can be found in Table 4.3 as well.

As Figure 4.4 and Table 4.4 show, the Random policy performed very well, in

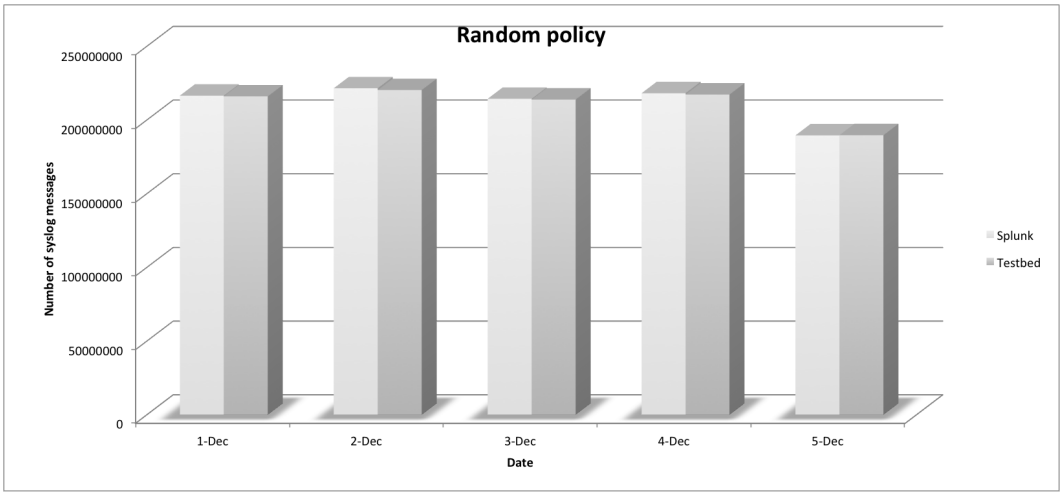


Figure 4.3: The Number of Syslog Messages After Filtering on the Testbed Compared to the Number of Messages Splunk Indexed Into Its Database for the Random Policy With No Server Failures.

terms of data delivery, in the second experiment as well.

4.2.3 Load-Based Policy

As well as the other policies, the Load-Based demonstrated a very good delivery ratio, with an average ratio of 1.03 and 1.00, as can be seen in Figure 4.5 and Figure 4.6 for the first and the second experiments, respectively. Numerical results can be found in Table 4.5 and Table 4.6, respectively. During the first experiment 0.96 and 1.10 are the lowest and the highest data delivery ratios, respectively. While during the second experiment these ratios are 0.96 and 1.04, respectively.

4.3 Load-Balancing Results

In this section we compare the different policies in terms of spreading the load among servers based on the results that we gathered from running them on the controller. We compare these policies for each experiment separately. To measure

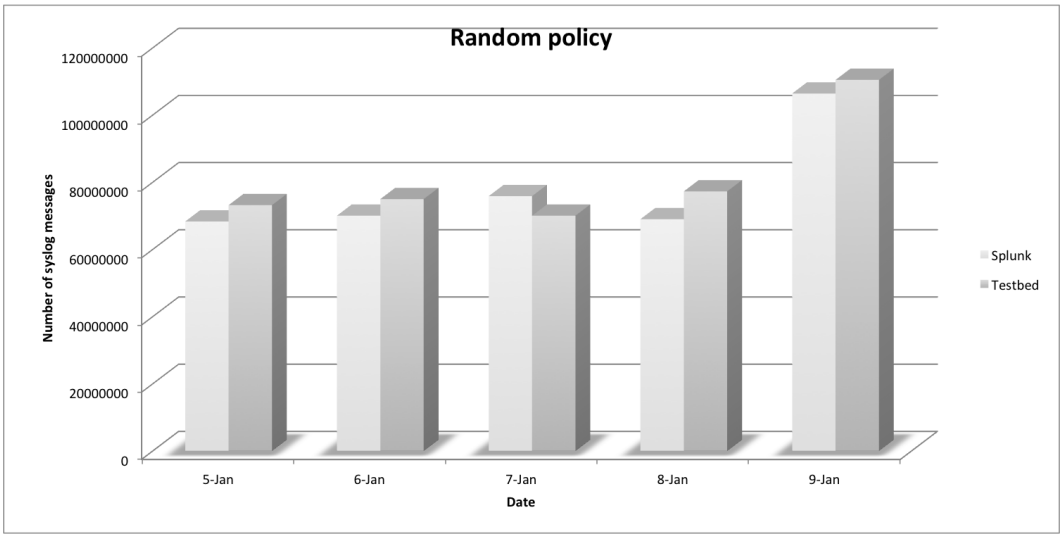


Figure 4.4: The Number of Syslog Messages After Filtering on the Testbed Compared to the Number of Messages Splunk Indexed Into Its Database for the Random Policy With Server Failures.

Date	Testbed (T)	Splunk (S)	Difference ($S-T$)	Delivery ratio (S/T)
Jan 5^{th}	73188622	68323257	-4865365	1.071210964
Jan 6^{th}	74993873	70059638	-4934235	1.070429068
Jan 7^{th}	70052433	75863551	5811118	0.923400396
Jan 8^{th}	77290725	69014981	-8275744	1.119912284
Jan 9^{th}	110467859	106417245	-4050614	1.038063511

Table 4.4: Total Number of Syslog Messages on the Testbed After Filtering Compared to the Number of Messages in Splunk Report for the Random Policy With Server Failures.

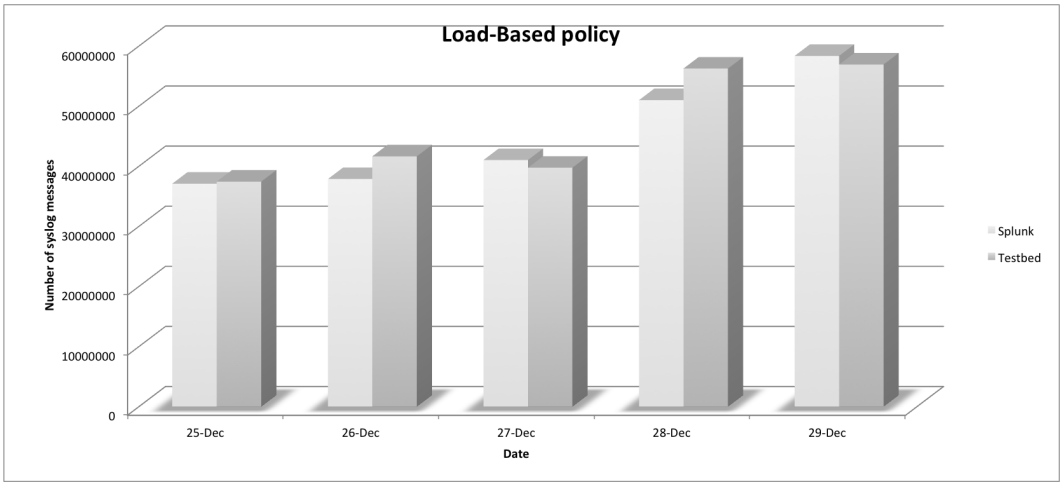


Figure 4.5: The Number of Syslog Messages After Filtering on the Testbed Compared to the Number of Messages Splunk Indexed Into Its Database for the Load-Based Policy With No Server Failures.

Date	Testbed (T)	Splunk (S)	Difference ($S-T$)	Delivery ratio (S/T)
Dec 25$^{\text{th}}$	37444009	37101507	-342502	1.009231485
Dec 26$^{\text{th}}$	41644757	37871078	-3773679	1.099645408
Dec 27$^{\text{th}}$	39752694	41029295	1276601	0.968885622
Dec 28$^{\text{th}}$	56260329	51003489	-5256840	1.103068243
Dec 29$^{\text{th}}$	56928490	58358202	1429712	0.975501096

Table 4.5: Total Number of Syslog Messages on the Testbed After Filtering Compared to the Number of Messages in Splunk Report for the Load-Based Policy With No Server Failures.

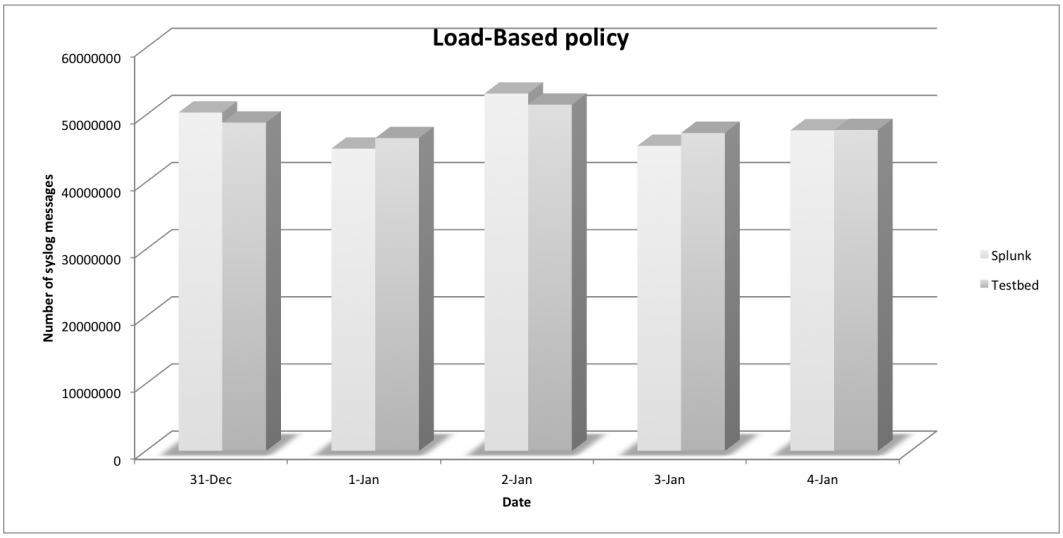


Figure 4.6: The Number of Syslog Messages After Filtering on the Testbed Compared to the Number of Messages Splunk Indexed Into Its Database for the Load-Based Policy With Server Failures.

Date	Testbed (T)	Splunk (S)	Difference ($S-T$)	Delivery ratio (S/T)
Dec 31$^{\text{th}}$	48887055	50388054	1500999	0.970211213
Jan 1$^{\text{st}}$	46580918	45021297	-1559621	1.03464185
Jan 2$^{\text{nd}}$	51567124	53219385	1652261	0.968953775
Jan 3$^{\text{rd}}$	47356854	45434582	-1922272	1.042308566
Jan 4$^{\text{th}}$	47786448	47741711	-44737	1.000937063

Table 4.6: Total Number of Syslog Messages on the Testbed After Filtering Compared to the Number of Messages in Splunk Report for the Load-Based Policy With Server Failures.

the performance of each policy in terms of spreading the load, we define the split ratio by dividing the smallest file size by the largest one stored on the testbed for each day.

4.3.1 First Scenario: Without Server Failure

In the first experiment all the servers are up and running for the whole experiment. This is the ideal network status that does not happen very often. We run all of the policies with the same configuration that might affect the result such as hard timeout for flows that we write on the switch. In this case the hard timeout for all policies is eight seconds. This means that after eight seconds the controller must select another server to forward the load.

The first policy we run on the testbed is Round-Robin. As Figure 4.7 and Table 4.7 show, Round-Robin divides the load among servers almost equally. The largest split ratio during this experiment is 0.99, while the smallest ratio is 0.98.

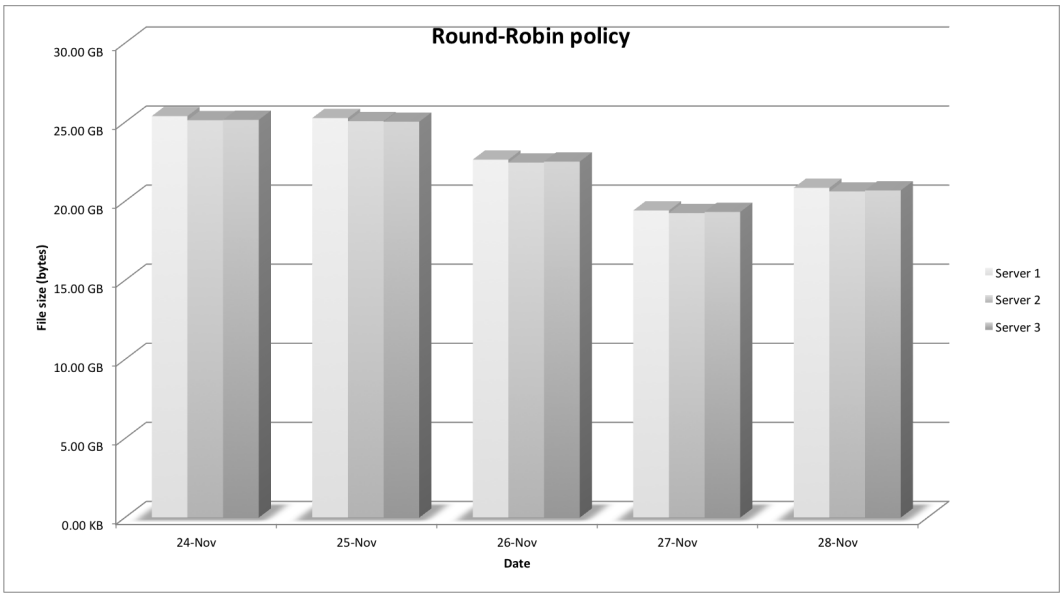


Figure 4.7: Round-Robin Policy Divides the Syslog Load Among Rsyslog Servers With No Server Failures.

Date	Server 1 file size	Server 2 file size	Server 3 file size	Split ratio (Min/Max)
Nov 24^{th}	25.38 GB	25.13 GB	25.15 GB	0.989912803
Nov 25^{th}	25.26 GB	25.06 GB	25.03 GB	0.990710266
Nov 26^{th}	22.63 GB	22.45 GB	22.51 GB	0.991811974
Nov 27^{th}	19.42 GB	19.26 GB	19.32 GB	0.991758144
Nov 28^{th}	20.85 GB	20.62 GB	20.68 GB	0.988974086

Table 4.7: Final Syslog File Sizes on Each Server at the End of Each Day When the Controller Was Running Round-Robin Policy With No Server Failures.

Date	Server 1 file size	Server 2 file size	Server 3 file size	Split ratio (Min/Max)
Dec 1^{st}	28.19 GB	28.19 GB	28.04 GB	0.99454847
Dec 2^{nd}	28.91 GB	29.24 GB	28.73 GB	0.982632883
Dec 3^{rd}	28.54 GB	27.51 GB	27.96 GB	0.964127122
Dec 4^{th}	29.24 GB	27.73 GB	27.84 GB	0.948370566
Dec 5^{th}	24.78 GB	24.98 GB	23.94 GB	0.958304583

Table 4.8: Final Syslog File Sizes on Each Server at the End of Each Day When the Controller Was Running Random Policy With No Server Failures.

Then, we run Random policy on the controller. The result is given in Figure 4.8. The Random policy also spreads the load in such a way that can be used in a network without any server failures. Table 4.8 shows file sizes at the end of each day on each server while the controller was running the Random policy. During this experiment the largest split ratio is 0.99, whereas the smallest ratio is 0.94.

Finally, we run the Load-Based policy on the testbed. As Figure 4.9 shows, the Load-Based policy behaved very similar to the Round-Robin policy. This policy spreads the load among the servers almost equally. Performance of all of the policies

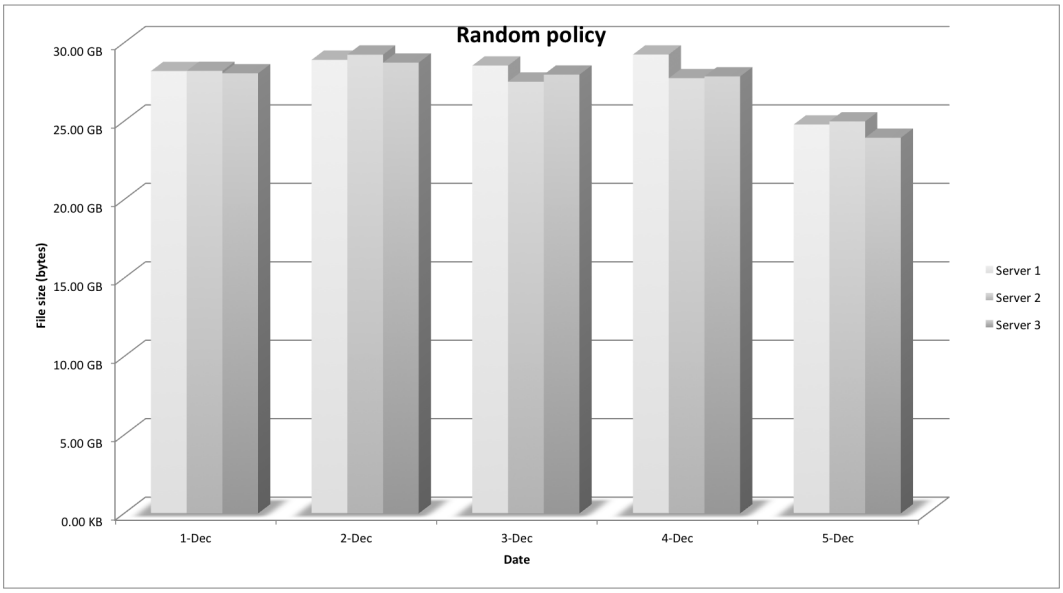


Figure 4.8: Random Policy Divides the Syslog Load Among Rsyslog Servers With No Server Failures.

was very good in terms of balancing the load between servers. Actual file sizes for this run can be found in Table 4.9. The largest and the smallest split ratios in this experiment are 0.98 and 0.97, respectively.

During the course of the first experiment the best average split ratio belongs to the Round-Robin policy by the rate of 0.99. While the Random policy has the worst average split ratio by the rate of 0.96.

4.3.2 Second Experiment: With Server Failure

In the second experiment we simulate the condition of losing one or two servers by turning off Rsyslog servers. Using XenCenter we reach the Rsyslog server that we want to turn off and shut it down. We examine the performance of all the policies in this condition, and then we compare their results.

First we run the Round-Robin policy. In Figure 4.10 we can see syslog file sizes

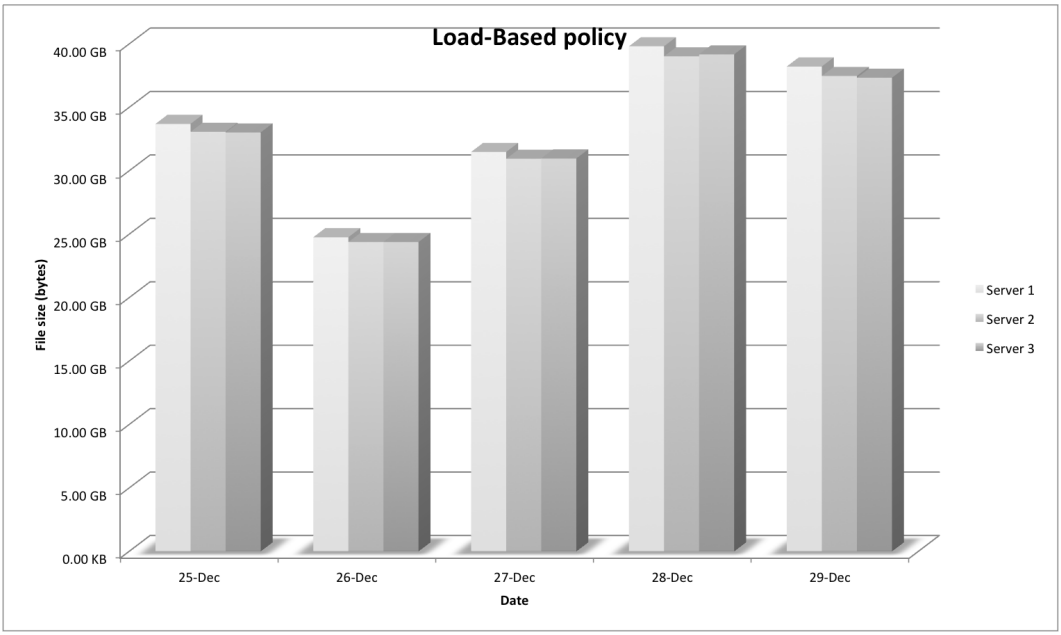
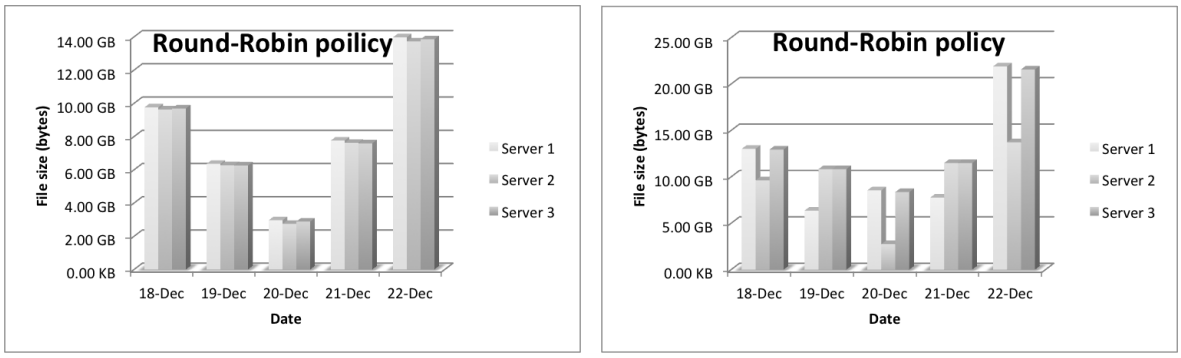


Figure 4.9: Load-Based Policy Divides the Syslog Load Among Rsyslog Servers With No Server Failures.

Date	Server 1 file size	Server 2 file size	Server 3 file size	Split ratio (Min/Max)
Dec 25^{th}	33.70 GB	33.07 GB	33.03 GB	0.980028907
Dec 26^{th}	24.76 GB	24.39 GB	24.38 GB	0.984667995
Dec 27^{th}	31.49 GB	30.95 GB	30.97 GB	0.98291316
Dec 28^{th}	39.81 GB	39.03 GB	39.18 GB	0.980243263
Dec 29^{th}	38.22 GB	37.48 GB	37.34 GB	0.976772638

Table 4.9: Final Syslog File Sizes on Each Server at the End of Each Day When the Controller Was Running Load-Based Policy With No Server Failures.

immediately before losing a server and at the time the server comes back up. Figure 4.11 shows the final syslog file sizes on each Rsyslog server at the end of each day; the actual file sizes are in Table 4.10. On average we lost one server for about two hours per day. In comparing Figure 4.11 with Figure 4.10b we see that this policy retains the gap created due to server failure between the server that experienced failure and the other servers. As a result the Round-Robin policy does not work well when servers fail. During this experiments the Round-Robin policy has the smallest and the largest split ratio of 0.75 and 0.90, respectively.



(a) Immediately before losing a server (b) Immediately after all servers come back up

Figure 4.10: Log File Sizes Immediately Before Losing a Server and Immediately After Server Recovery.

Now, we explore performance of the Random policy. As Figure 4.12 depicts, after the server recovers, the gap in the file sizes is retained. But as we have seen in Round-Robin policy, the Random policy is not able to fill this gap after recovery. The Random policy obtains the smallest and the largest split ratio of 0.80 and 0.97, respectively.

As Figure 4.13 shows, at the end of each day we can see that the gap remains. Hence the Random policy is not suitable when servers fail. Table 4.11 shows the final syslog file sizes for this period.

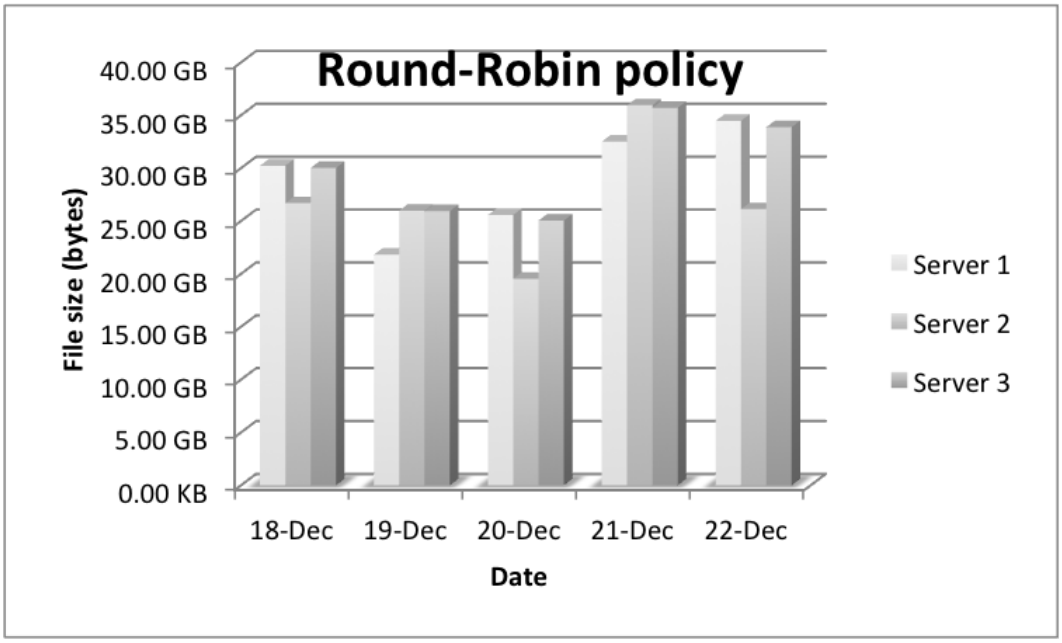
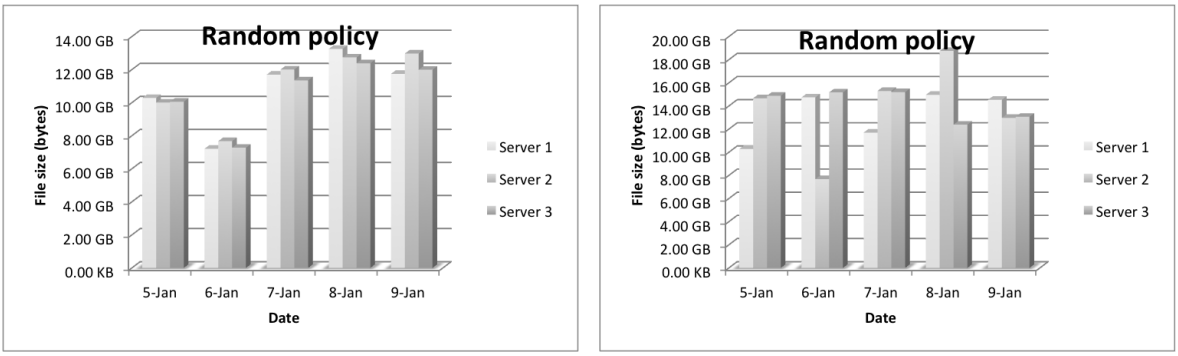


Figure 4.11: Syslog File Sizes on the Testbed at the End of Each Day for the Round-Robin Policy With Server Failures.

Date	Server 1 file size	Server 2 file size	Server 3 file size	Split ratio (Min/Max)
Dec 18th	30.23 GB	26.68 GB	30.02 GB	0.882603893
Dec 19th	21.82 GB	25.98 GB	25.94 GB	0.840052407
Dec 20th	25.57 GB	19.54 GB	25.04 GB	0.764244724
Dec 21th	32.50 GB	35.95 GB	35.71 GB	0.90421896
Dec 22th	34.47 GB	26.12 GB	33.87 GB	0.757887583

Table 4.10: Final Syslog File Sizes on Each Server at the End of Each Day While the Controller Was Running Round-Robin Policy With Server Failures.



(a) Immediately before losing a server (b) Immediately after all servers come back up

Figure 4.12: Log File Sizes Immediately Before Losing a Server and Immediately After Server Recovery.

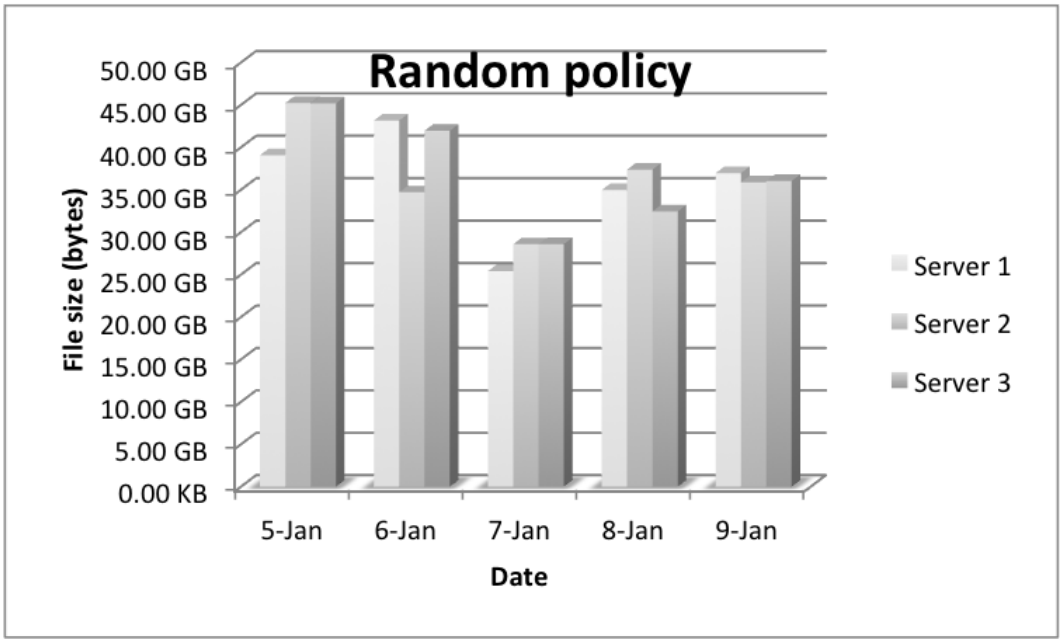
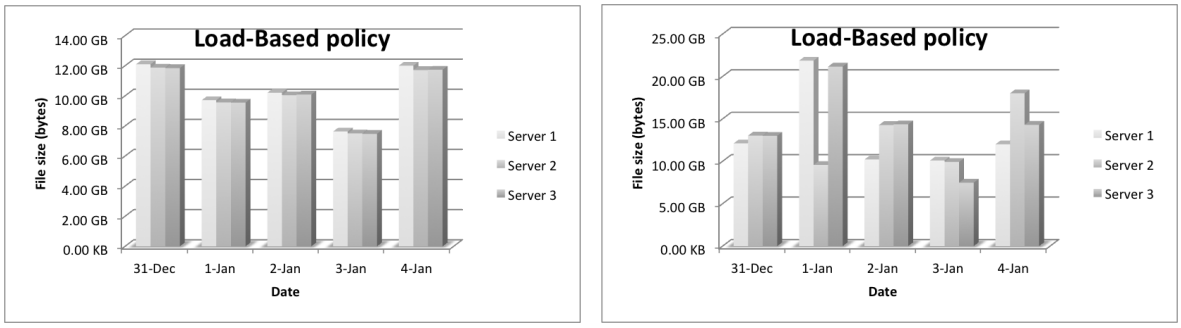


Figure 4.13: Syslog File Sizes on the Testbed at the End of Each Day for the Random Policy With Server Failures.

Date	Server 1 file size	Server 2 file size	Server 3 file size	Split ratio (Min/Max)
Jan 5^{th}	39.07 GB	45.23 GB	45.18 GB	0.863902131
Jan 6^{th}	43.16 GB	34.70 GB	41.95 GB	0.80407031
Jan 7^{th}	25.40 GB	28.57 GB	28.59 GB	0.888571795
Jan 8^{th}	34.98 GB	37.37 GB	32.44 GB	0.868200106
Jan 9^{th}	36.96 GB	35.89 GB	36.03 GB	0.970871603

Table 4.11: Final Syslog File Sizes on Each Server at the End of Each Day While the Controller Was Running Random Policy With Server Failures.

Finally, we run Load-Based policy on the controller. In Figure 4.14 we can see syslog file sizes before the server fault and after server recovery.



(a) Immediately before losing a server (b) Immediately after all servers come back up

Figure 4.14: Log File Sizes Immediately Before Losing a Server and Immediately After Server Recovery.

As can be seen in Figure 4.15 and Table 4.12 this policy is able to compensate for the time that some of the servers were down (unless, the server was to fail at the end of the day and there is insufficient time for recovery). This policy tries to connect the imbalance in file sizes due to a fault network or server. The behavior of this policy is very similar to the Round-Robin and Load-Based policy in the first

experiment where we had all the servers up for the whole experiment. During this experiment the Load-Based policy performs the best among all of the policies with the largest and the smallest split ratios of 0.97 and 0.98, respectively. As a result, we can conclude that the Load-Based policy is the most appropriate for a network where servers may fail.

During the second experiment, the Round-Robin policy obtains the worst average split ratio with the ratio of 0.82. Whereas the best performance in terms of load-balancing belongs to the Load-Based policy with the split ratio of 0.97.

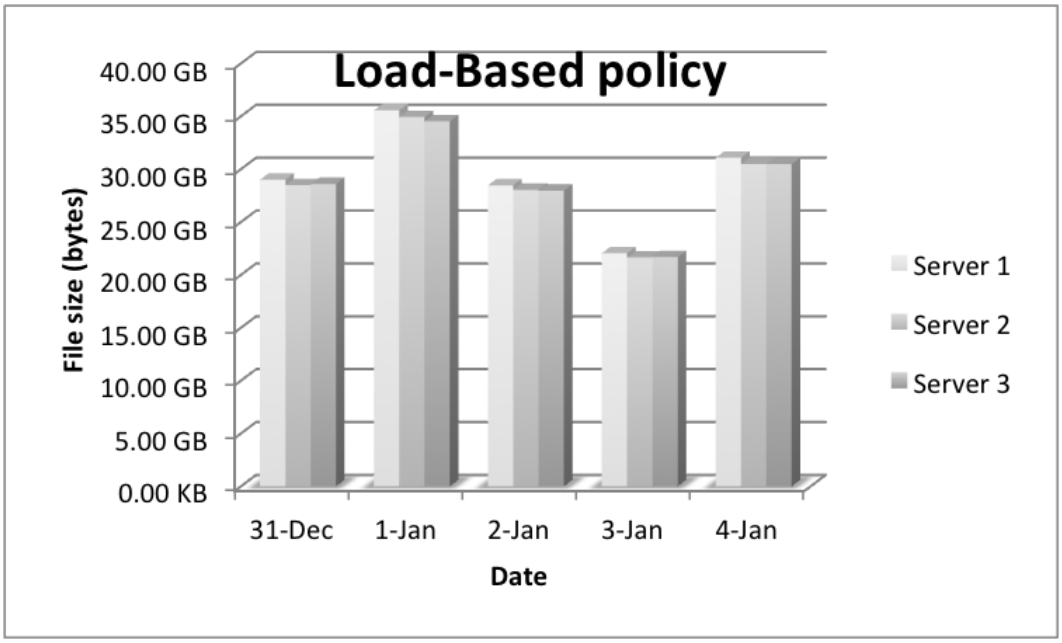


Figure 4.15: Syslog File Sizes on the Testbed at the End of Each Day for the Load-Based Policy With Server Failures.

4.4 Overall Summary

Our evaluations on different SDN-based transport-level load balancing policies suggest that we can divide the load almost equally among Rsyslog servers using any policy when server failures are unlikely to happen. We also demonstrated that using a

Date	Server 1 file size	Server 2 file size	Server 3 file size	Split ratio (Min/Max)
Dec 31st	29.00 GB	28.48 GB	28.60 GB	0.982225926
Jan 1st	35.55 GB	34.92 GB	34.52 GB	0.970855028
Jan 2nd	28.47 GB	28.02 GB	27.96 GB	0.982183892
Jan 3rd	22.04 GB	21.64 GB	21.67 GB	0.981745024
Jan 4th	31.07 GB	30.48 GB	30.48 GB	0.98101208

Table 4.12: Final Syslog File Sizes on Each Server at the End of Each Day While the Controller Was Running Load-Based Policy With Server Failures.

Load-Based policy is most appropriate to divide the load almost equally when servers may fail. With overall average delivery ratio of 1.01 and the split ratio of 0.92, our results suggest that SDN-based load balancing could replace the existing solution.

In the next chapter, we summarize our contributions and propose future work.

Chapter 5

CONCLUSION AND FUTURE WORK

In this thesis, we discussed the need for an SDN transport-level load balancing solution over UDP packets. We discussed three different load balancing policies and how we can take advantage of SDN specifications and use network statistics that it offers to perform a network-aware load balancing over UDP packets. We implemented three different policies and we tested them in two experiments in our testbed with a data feed from ASU's production network.

We defined delivery ratio as a metric to validate whether our solution is reliable. This ratio is calculated by dividing the number of the syslog messages received on the testbed in a day to the number of the syslog messages indexed by Splunk for the same day. The average delivery ratio for the Round-Robin, Random, and Load-Based policies during the first experiment, without having server failures, is 0.99, 0.99 and 1.03, respectively. Since the path to our testbed is different from path going to ASU's servers, and because the data is being sent in UDP, there is no guarantee that we get exactly the same data as ASU's Rsyslog servers. As a result the delivery ratio can be more than one. During the second experiment, with simulation of server failure, the average delivery ratio for the Round-Robin, Random, and Load-Based is 0.99, 1.04 and 1.00 respectively. The results of these experiments suggest that these SDN load balancing policies are reliable enough to be used in a real production network with a high input data rate. The results show our solution is competitive with the existing commercial load-balancing solution for UDP traffic.

A split ratio is a metric defined to measure the ability of each policy to divide the load equally among the servers. Dividing the smallest sized file on the testbed for a

given day by the largest sized file on the testbed for the same day, gives us the split ratio of that specific day. In the first experiment average split ratios for the Round-Robin, Random, and Load-Based policies were 0.99, 0.96 and 0.98 respectively. The split ratio for different policies during the first experiment is 0.99, 0.96 and 0.98 for the Round-Robin, Random, and Load-Based policies, respectively, and Round-Robin had the best ratio among them all. But for the second experiment the results were quite different. The average split ratio over the period of the second experiment is 0.82, 0.87 and 0.97 for the Round-Robin, Random, and Load-Based policies, respectively. The Round-Robin and Random policies could not divide the load during the second experiment as equally as they did in the first experiment. However, the Load-Based policy's performance during the second experiment showed it could recover from server failures.

In summary, the results on split ratios indicate that while each load balancing strategy is effective when there are no server failures, when servers fail, only the Load-Based policy can recover and keep the split ratio high. Therefore, if equal file size is an outcome of load balancing, the Load-Based policy is most appropriate.

In this project we used an OpenFlow enabled software switch to balance the load. Future work could use an OpenFlow enabled hardware switch to perform the load balancing. This should only improve our results because switching rates are faster on physical hardware and would even allow us to handle input at a faster data rate.

In this thesis we restricted our attention to balance the load over the UDP protocol. The Palo Alto firewall also can run over TCP. By using TCP instead of UDP, future work can eliminate loss of packets. However, this change complicates the controller design since connection setup and state management would need to be maintained across the servers.

REFERENCES

[1] Floodlight. http://www.projectfloodlight.org/floodlight/.

[2] Mininet. http://www.mininet.org/.

[3] Open vSwitch. http://www.openvswitch.org.

[4] OSGi. http://www.osgi.org/Technology/WhatIsOSGi.

[5] OSGi Bundle. http://www.osgi.org/javadoc/r4/org/osgi/framework/Bundle.html.

[6] Splunk. http://www.splunk.com/resources.

[7] Big Switch Networks. Open source projects supported by Big Switch Networks. http://bigswitch.com/products/open-source-projects.

[8] Valeria Cardellini, Michele Cola Janni, and Philip S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[9] Citrix. Citrix NetScaler. http://www.citrix.com/products/netscaler-application-delivery-controller/overview.html.

[10] Citrix. XenCenter. http://xenserver.org/open-source-virtualization-download.html.

[11] Citrix. XenServer. http://www.xenserver.org.

[12] Dipjyoti Saikia, SeokHwan Kong, Nikhil Malik, Dayoung Kim. OpenMUL. http://www.openmul.org.

[13] Advait Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. Towards an elastic distributed SDN controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 7–12. ACM, 2013.

[14] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pages 407–416, New York, NY, USA, 2000. ACM.

[15] R. Gerhards. The Syslog Protocol. RFC 5424, Internet Engineering Task Force, March 2009.

[16] Global Environment for Network Innovations (GENI). http://groups.geni.net/geni/wiki/GeniNewcomersWelcome, 2010.

[17] Gordon Lyon. Nmap. http://www.nmap.org.

[18] Nikhil Handigol, Mario Flajslik, Srini Seetharaman, Nick McKeown, and Ramesh Johari. Aster*x: Load-Balancing as a Network Primitive. In *Architectural Concerns in Large Datacenters (ACLD'10)*, 2010.

[19] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, CoNEXT '12, pages 253–264, New York, NY, USA, 2012. ACM.

[20] Nikhil Handigol, Srinivasan Seetharaman, Mario Flajslik, Nick McKeown, and Ramesh Johari. Plug-n-serve: Load-balancing web traffic using openFlow. SIGCOMM Demonstration, 2009.

[21] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communications*, 38(2):69–74, 2008.

[22] Open Networking Foundation. OpenFlow switch specifications. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow, 2009.

[23] J. Postel. User Datagram Protocol. RFC 768, Internet Engineering Task Force, Augest 1980.

[24] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. FlowVisor: A Network Virtualization Layer. Technical report, Deutsche Telekom Inc. R&D Lab, Stanford, Nicira Networks, 2009.

[25] The Apache Software Foundation. Maven. http://maven.apache.org/what-is-maven.html.

[26] The Linux Foundation. OpenDaylight controller. http://www.opendaylight.org/announcements/2013/04/industry-leaders-collaborate-opendaylight-project-donate-key-technologies.

[27] Hardeep Uppal and Dane Brandon. OpenFlow Based Load Balancing, 2010.

[28] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Proceedings of USENIX*, 2011.